Table of Contents

4 Self-Modifying Systems

This chapter lays the foundation for the analysis of strong AI as metamorphic software. It provides an introduction to the relevant concepts surrounding self-modifying systems, as applied to computer science. These hardware and software computing systems are capable of rewriting or reconfiguring their architecture. This is relevant to AI security and safety because such systems have the ability to manipulate descriptions with effectiveness, affording them the potential to evolve beyond their original limitations or specifications. In addition to rewriting their own program descriptions, they will also be capable of rewriting other data and program descriptions accessible to them. This includes the ability to penetrate logical boundaries, such as operating system calls, application programming interfaces, and hardware interfaces. The implications of this will be their ability to self-replicate through our global network infrastructure, cross hardware and software boundaries, evade detection, and overcome countermeasures.

4.1 Codes, Syntax, and Semantics

A code is a method or scheme that specifies how to convert information into different forms or representations [1, 2, 3]. To put it into terms consistent with previously discussed concepts, this would mean that a code describes a method for transforming descriptions, either within the same description language or into that of another description language entirely. As a result of this, codes can be interpreted as the syntax for some description language.

The word code can also be used to refer to the entirety of a description, e.g. “source code”. The distinction is usually inferred through context, but it will be made explicit here for clarity by referring to it as a code scheme.

A formal language [4, 5, 6] is defined as a set of words over an alphabet. The words must be finite, but the language itself may be infinite. This formal definition makes mention of neither syntax nor semantics. It is simply that set, constructed in that particular way, with nothing extra implied. It is exactly that and nothing other than that unless something is explicitly added or attached to it in the relevant context. For reasons of brevity and practicality, the description languages mentioned in this book are restricted subsets of formal languages that have been constrained to *working* descriptions called *implementations*. This is because formal languages include nonsensical descriptions, e.g. “colorless green ideas sleep furiously” [7]. To be clear and concrete, these qualifications define these description languages necessarily as subsets of the formal languages that entail them, i.e. the formal language for a given description language is possibly (infinitely) larger, but contains descriptions that are disregarded for practical use.

Importantly, it is the semantics, not the syntax, that determine the power of languages [8, 9, 10] and the corresponding computational extents of description languages. To understand this, we must first see how formal grammars are distinct from formal languages. One can generate some formal language from a particular formal grammar, but, as mentioned above, formal languages have neither syntax nor semantics as part of their formal definition. Further, the ability for a restricted class of abstract machines to recognize certain grammars and not others [11], as shown in the Chomsky hierarchy [7, 12, 91, 92], does not imply restrictions in the descriptions it generates. There exist Turing-complete instruction sets [13] and programming languages with grammars above the unrestricted class of grammars. This might seem confusing at first, but can be made clearer by accepting a full disconnect between grammar and language with respect to efficacy or power. This is due to the fact that all languages must be interpreted by some process in order to have any effect [14]. In terms of programming languages, this is by means of explicit computation that implements, or reifies, the *semantics* of the language [15, 16].

While formal languages do not include syntax or semantics, they can be qualified to sets of working descriptions very easily through the intension definition [17] of sets, e.g. “the set of all working AI implementations.” By contrast, a formal grammar requires a definition in the form of production rules that recognize or generate some language; this would potentially require enormous numbers of productions [18, 19, 20], and, in some cases, may be unknowable for practical reasons. This hints at the subtle relationship between conventional machine learning and the limits of narrow AI. In other words, without the relevant semantics at hand, one degenerates to the brute-force enumeration of countless permutations, all the while never having the capacity to effectively apply those rules outside of the context in which they were derived by rote calculation. This is because the problem of semantics is vastly more involved than the mere juxtaposition [21] of symbols, especially when those symbols have meaning. In the context of computational languages, this meaning comes in the form of functionality that must be captured in an implementation, a description that entails the semantics.

4.2 Code-Data Duality

Computers are primarily programmed through the transformation of human readable source code into descriptions that can be directly and indirectly executed by the machine. Compilation is the process of transforming that source code from one description language into that of another. This typically results in native machine code in the description language of some microprocessor or microcontroller. In other cases, the result of the compilation is generated in a description language that differs from the native description language of the machine and must be interpreted to be executed. Finally, source code may be interpreted directly, with or without a corresponding compilation process.

The underlying hardware is always involved, it’s just a question of how many layers of abstraction are between the given description and the machine’s native description language, called the instruction set. It is possible to nest, encode, or embed descriptions within descriptions ad infinitum. The limits of this are determined by each description language and the resource constraints of the implementation.

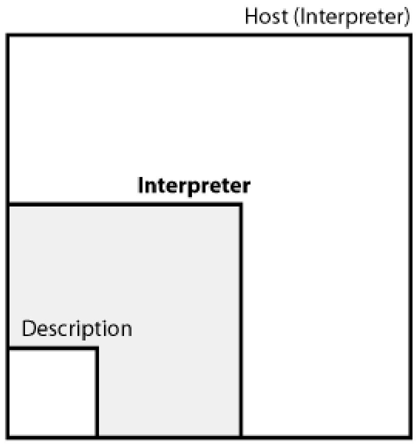
In all cases above, the descriptions are data being interpreted by some process, with that process being either another program or the machine itself. This is the duality between code and data, and it’s the basis of self-modification in computing. This duality helps to unify the notion of interpreter and machine in the interpretation of information, which is what is being referred to when one discusses information exchange. In other words, information exchange is not possible without an interpreting process, and every interpreting process must have syntax and semantics as part of its construction, otherwise it would be incapable of signification.

A description language with semantics that provide access, recognition, and generation of its syntax and semantics *while being interpreted* is called *reflexive* [22, 23]. However, this property is not required for self-modification in general. This is related to the class of self-interpreting languages that exhibit homoiconicity [24], which represent the implementation details of the description language in terms of the structure of the language itself. This makes these languages trivially reflexive. In all cases, however, the limits of self-modification are up to the semantics of the interpreting process and not particular language features, which only make it more or less convenient to perform self-modification.

4.3 Interpreters and Machines

An interpreter is an implementation that evaluates descriptions written in a description language [25]. If in software, the interpreter would be either a program or embedded as part of another program. In hardware, it would be a microprocessor or microcontroller or, potentially, some customized integrated circuit or other hardware. The language that the interpreter hosts may be either the same or different from the one it was implemented in itself.

Figure 4.1: Conceptual construction of interpreters as nested implementations of description languages.



Each box represents a single description out of possibly many descriptions entailed by the implementation that contains it. Every interpreter is a description in some description language that also represents an implementation of a (possibly different) description language. This continues schematically towards the upper right diagonal ad infinitum until reaching discrete physical descriptions.

Every interpreter follows the construction above: a description in a host description language that implements another, possibly distinct, description language. The power of the interpreted language is always determined by the semantics of the implementation. This can be applied to nearly any context that permits a physical description of the system.

In this book, all interpreters are generalized to a category or class under the above construction. This confers the advantage of a universal analysis that provides broad coverage of the relevant principles of self-modification. This generalization includes all of the types of interpreters that will be discussed in this section, both hardware and software alike, and should not be confused with the light-weight interpreters that are about to be discussed.

Often, in the context of computer programming, interpreter typically refers to a specific sub-class of interpreters that utilize abstractions (or no abstractions) that are relatively close to the form or function of the description language they host. These interpreters either use some internal representation or perform direct execution as the language is parsed. In some cases, these interpreters will utilize what is known as byte-code [26], which is a compiled version of the human readable source that has been put into a description language that is more efficient for machine reading. This byte-code is usually not in the language of the host architecture that runs the interpreter, and must be read by the interpreter itself.

Just-in-Time (JIT) compilers combine the features of the light-weight interpreters above and that of compilation by allowing descriptions to be analyzed and transformed at runtime [27].

A virtual machine (VM) is a type of interpreter that goes a step further and implements, or, more correctly, emulates, the semantics of a particular computing architecture or platform [28]. This creates an additional level of abstraction or indirection between the description language being hosted and the underlying architecture that implements the VM. The benefits of this is that the VM can trade relative performance for platform and hardware independence [29, 30]. It also admits the possibility of unique security features, as it is not compelled to execute every program or series of instructions that it encounters [31, 32, 33]. Hardware circuitry could be implemented to do the same [34, 36, 37, 38, 39, 40] but it may not be as flexible or dynamic as a VM. Ultimately, however, a VM is still an implementation, and any security at this level would still be self-security, and subject to fundamental vulnerabilities [41]. The VM concept primarily gains attention due to its ability to run descriptions on multiple architectures without recompilation.

Virtualization takes the VM concept a step further by creating one or more VMs across computing systems of potentially different types of architectures [42]. This can be used to split computational resources into smaller units or combine them to form a larger logical machine.

Finally, a microcontroller or microprocessor is an interpreter implemented as hardware. It has an instruction set which exposes hardware semantics at the lowest levels. It is at this level that other interpreters are implemented, including VMs, which must have an associated native implementation portion to enable the emulation of the architecture.

All interpreters have a set of advantages and disadvantages that are determined by their descriptions. There can be no general purpose description that is optimal for every type of algorithm or program. This truism is the result of the semantics of the implementation and the levels of abstraction between it and the discrete physics. Every layer of abstraction or analogy induces another level of interpretation that must be performed. As such, the most efficient descriptions are those that minimize these levels of indirection [43, 44, 45, 46, 47, 48]. A digital computer is still an analog device in that it utilizes electrical states as representations for digital logic. Its storage mechanisms manipulate and interpret physical states which are analog representations constrained to specific ranges of interpretation.

The optimal computer for a given problem is one in which there is as close to a one-to-one correspondence between the algorithm being implemented and the physical analogues used in its interpretation. This is perhaps why the human brain is such an efficient computational system, even without factoring in its ability to reconfigure itself as it undergoes learning.

Further issues that confound any general purpose solution is that certain problems are recalcitrant to parallel processing [49, 50, 51]; some can benefit from the ordering of instructions and the prediction of branching in logic; others are benefited by cache in terms of memory and instructions, while others would need bigger memory bandwidth, as they do not benefit from cache as well as others due to their dynamics. The bottom line is that there are compromises in the design of any interpreter that are often decided by the context in which they will be used most frequently. This is another reason why it is almost always nonsensical to discuss in the absence of specific implementations.

This is all related to self-modification in that the above limitations are exactly why a system would benefit from the ability to self-modify. These are non-trivial problems that are unavoidable and recalcitrant. Take Amdahl's law, for example, which gives a relationship between the maximum speed-up that can be attained by adding more processors to a problem with even a fractional serial portion [52]. That is to say, any given problem can be represented by how much of it is necessarily serial in its execution. As mentioned above, not all problems benefit from parallel computation.

Often, the most difficult part that professional software engineers and applied mathematicians have to struggle with is finding the optimal algorithms constrained by the relevant semantics of the description languages given to them. A professional graphics developer is a perfect example: they have cutting edge, specialized physical hardware descriptions which provide unique semantics, but these semantics are never as flexible as just specifying a physical implementation of the algorithm. This is because these systems have to fit within a manufacturing process and made general enough to serve a wide variety of applications; they are specialized, but not so specialized that they would only calculate a singular set of problems.

In summary, an interpreter is always required for any description language to have effect. At its most fundamental, the physical world can be thought of as an interpreter with what we call reality being its descriptions. This is not to advertise a digital physics, as descriptions need not be represented digitally! It is to say that this concept of an interpreter is almost universal, depending on how willing one is to relax their conceptual boundaries. Every interpreter has to implement the semantics and recognize the syntax of the description language they host, with the power of that language being determined by the semantics of the implementation. The implementation's efficiency is determined partially by its syntax and semantics, plus the levels of abstraction between it and that of some discrete physical description.

Every underlying physical interpreter involves a form of analog computing or representation, and the most efficient possible forms of computation would be those which are as close to a one-to-one correspondence with this as possible. This raises the fundamental trade-off between generality and specialization, in that the closer we arrive to this one-to-one correspondence, the more constrained the implementation becomes for a particular set of problems. This sets the bounds of the problem space for which self-modification is applied, as it seeks to have both generality and specialization through the self-modification process.

4.4 Types of Self-Modification

All systems capable of self-modification share a few basic operations: rewrite, replicate, and generate. These are either direct or indirect in operation. A description (implementation) is in either one of two states: online or offline.

Offline descriptions are the conventional type of description that have been discussed so far about implementations. Recalling the time-like objects and processes from the previous chapter, offline descriptions entail potential states and operations and are not the physical description of an active implementation. They are an entailment of its potential. By contrast, an online description entails the *relevant* operational and physical information of an implication but only insofar as it is required to undergo self-modification. This may include state and environment information or anything else that would be required to complete self-modification under *active* operation.

Direct self-modification is where the system is applying modifications to itself without external sources for aid. Indirect self-modification is for cases where the system utilizes external sources to self-modify, but with the stipulation that it must have either created or initiated these sources. A supervised self-modification is still self-modification as long as it is able to operate directly or indirectly upon itself without further assistance. Intervention, cessation, or interruption of this process by a supervising process would not constitute aid unless it qualitatively alters the self-modification process beyond the starting or stopping of its execution.

A summary of the types of self-modification:

- Direct
  - Rewrite
  - Replicate-Rewrite-Replace
  - Generate-Replace
- Indirect
  - Replicate-Rewrite-Replace
  - Generate-Retroactive Rewrite

Rewrites involve partial or complete modification of the original description. There is no implied copying. Rewrites may apply to descriptions which are either online or offline. An offline description is simply a copy of a system's description which is no longer considered part of it. An online implementation that engages in rewrite self-modification must be capable of handling live edits of its description without generation or replication. This lack of copying is what distinguishes rewrite as a basic operation in self-modification from replication and generation. Thus, when rewrite occurs to an offline description it is necessarily an intermediate stage towards self-modification, as it will require an additional step to be considered part of the original system once more.

Replication is the *unmodified* copying of a system's description. It is an intermediate step towards self-modification. This replication is capable of occurring within a system without instantiating the copied description. It may be a temporary working copy or part of the process of self-modification. Replication does not imply an online description, but it does imply *replacement* if it is to be considered part of any self-modifying process.

Generation can be thought of as a combined replicate and rewrite process, but is distinct from both as it is the direct construction of a description from a process. This is to be contrasted with the replication, rewrite, and replace method of self-modification, which involves the modification of a copy. The defining characteristic of generation is that it involves computation, and, as such, can be generalized as a kind of compression, in which the generative system has an effective "copy" of all the possible permutations it can potentially create in an ultra-compact representation. Like replication, generation implies an eventual *replace* to be considered self-modification. This is because generation creates a separate description and does not modify the original system (such a case would be a rewrite).

Retroactive replacement is for the complex case where a system is, for some reason, incapable of direct self-modification. As a result, it generates or replicates a description, which may or may not need to be modified further, and then utilizes that as a means of self-modification.

Finally, there needs to be a discussion regarding *iteration*. Why would this not be considered self-modification? The reason is that it does not imply a change in the identity of the originating process and that it does not serve the same function or purpose of self-modification, which is to break through description language barriers (to be discussed ahead).

4.5 Reconfigurable Hardware

Application-specific integrated circuits (ASIC) are customized integrated circuits designed for a specific functionality and purpose [53]. They range from partial to fully custom designs that can vary greatly in performance and cost. The benefit of ASICs are their high performance and lower marginal cost at high production volumes [54]. The drawback is that they can not be reconfigured once manufactured.

By contrast, and of relevance to self-modification, is the field programmable gate array (FPGA), which uses an array of programmable logic blocks that allow significant changes to a circuit design, and can be re-configured and reused for many applications [55, 56, 57]. Limitations include only partial update during operation [58, 59], a significantly more complex reprogramming process than software [60], and volatile storage of the configuration [61]. The latter issue of volatility can be overcome with system-on-a-chip designs that include common integrated circuit (IC) components as part of the FPGA itself [62, 63, 64]. This hardware is commonly used to prototype and verify ASIC designs [65, 66, 67] and can potentially be more cost effective than an ASIC, at low volumes, even for production use [68].

Firmware is defined as the combination of integrated circuits (ICs) with non-volatile memory to allow logic to be represented as software directly in the hardware [69]. This provides some flexibility and modularity to the hardware but not to the same extent as an ASIC or FPGA. This is because the stored software is still forced to operate on a specific circuit and data-path configuration, and it can not be altered unless combined with one of the above configurable devices.

At the time of this writing, there exists no configurable hardware technology that compares with the freedom and ease of self-modification available to software. The trade-off, however, is run-time efficiency. This means that while software is capable of virtually unrestricted self-modification in terms of algorithmic creativity, it comes at the cost of being limited (in efficiency) by the semantics of the hardware that serves it. This could be limiting in cases where solutions benefit significantly from parallelization, but it is important to note that this does not fundamentally prevent the ability to run these implementations at a less optimal pace. A slow strong AI could potentially be significantly more effective than even the brightest human.

Reconfigurable hardware, including FPGA, firmware, and embedded systems also have their own set of unique security challenges [70, 71, 72, 73, 74, 75, 76, 77, 78, 79]. It is because of this, and the above benefits of software, that the focus of this book is not on hardware, but software. The reasons are clear: no current hardware technology has the ability to efficiently and effectively self-replicate or modify at sufficient rates. Software does, and, with the increasing ubiquity in connectivity, it has the means to spread very rapidly. Thus, the rest of this chapter will focus on the software aspects, which follows more closely inline with the foundations presented in the previous chapter on AI implementations.

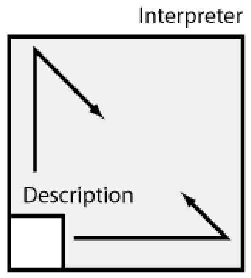
This does not mean that we should ignore the physical parts of implementations or the vulnerabilities in hardware. It is only to say that, in the context of self-modification, the path of least resistance will be software. That this is the route most likely to appear first and the one to be taken most easily by malicious users. It will also be the most difficult to restrict, enforce, and contain; properties that make it a prime medium for misuse.

4.6 Purpose and Function of Self-Modification

Descriptions are incapable of transcending their respective description languages without transformation. This is the fundamental barrier that self-modification seeks to overcome. If we want an optimized set of semantics or a more efficient syntax, we have to have different descriptions. To get from one description to another requires an explicit process. Self-modification occurs when this transformation takes place within a single identity and the entity at the source and destination are (eventually) the same.

As was defined above, the description languages discussed in this book pertain to working, well-formed, possible descriptions, along with their relevant syntax and semantics. As a result, an interpreter can be seen as an implementation of a description language. In such a case, we are referring not to the description of the interpreter but to that of the possible programs of the description language it entails. This brings us to the actual focus of this section:

Figure 4.2: Language barriers as fundamental limits in self-modifying systems.



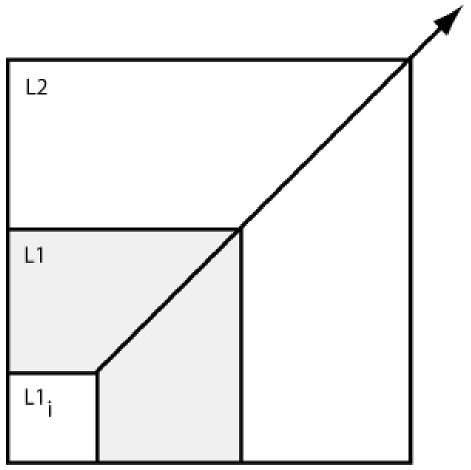
Any description is contained within the description language implemented by an interpreter unless there are enabling semantics to transcend it. This applies universally to any physical description that can undergo (self-)modification.

In the figure above, each box represents a description. The description undergoing self-modification is the smallest, innermost box in the lower left. It represents only a single instance out of many possible descriptions entailed by the interpreter. What this shows us is that the self-modifying implementation depicted is capable of attaining any possible description in the interpreter's set of possible programs. This is a trivial form of self-modification, as it is still locked within the description language of the interpreter, and of the description language that implements that interpreter, ad infinitum. In practice this is never infinite, being bound by practical limits and real-time demands. A discrete physical description is attained relatively quickly, semantics permitting.

This first trivial form of self-modification illustrates the line of demarcation between a self-hosting interpreter and a meta-circular interpreter. Typically, a self-hosting interpreter implements its description language semantics directly. This is neither unusual nor interesting as a point itself; however, when contrasted with meta-circular interpreters, it becomes a limitation. This is because the description is locked within a self-interpreter's framework. Meta-circular interpreters (only partially) overcome this by exposing the underlying interpreter that hosts it [80, 81]. This makes lower level semantics part of the semantics of the description language, and, as a result, effectively makes the interpreter transparent and reflexive. This allows for the meta-circular interpreter to implement new description languages by building on the primitives of its underlying host semantics. It may seem an ideal candidate for self-modification, but this too is still limited when compared with the next stage.

The most powerful form of self-modification is where the interpreter can be bypassed entirely for the next stage of interpretation:

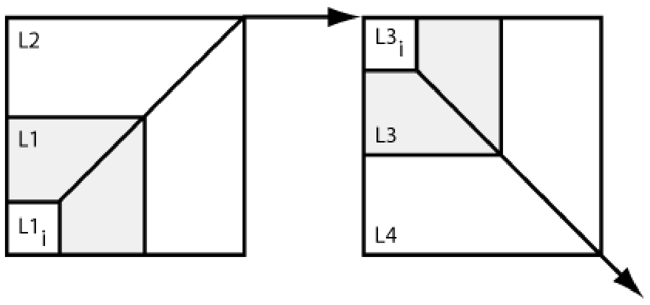
Figure 4.3: Self-modification transcending description language barriers.



The diagonal represents what the self-modifying description (lower left) is able to achieve with enabling semantics. Bypassing the interpreter **L1**, it is free to penetrate levels of abstraction, including **L2** and beyond, until a semantic barrier or a discrete physical description is reached.

In the above figure, the self-modifying program is breaking through its original interpreter and accessing the interpreters at each successive stage. Unlike the light-weight interpreters previously described, it has the ability to completely escape each description language boundary. That is, it is activating and bypassing levels of abstraction in terms of interpreters, which are implementations of description languages. It can do this until it reaches the level of the discrete physics, where, hypothetically, it would only be limited by the material and energy resources available to it through some form of physical reconfiguration. Note that this is beyond the level of reconfigurable circuits, and would require nanotechnology or an equivalent process that can manipulate the physical description of the computational substrate. This diagonal escape from each description language is possible as long as the semantics exist to access the next lower level at that stage in the process.

Figure 4.4: Self-modification transcending description language barriers through peripheral or network access.



This process could be achieved through any means which enables the self-modification process. This would still be limited by enabling semantics and the required knowledge of the destination description languages and interpreters.

If a limitation is reached where there are no longer any semantics that implement next lower-level access then it is possible for the self-modifying system to escape that meta-stable state by accessing network or peripherals. This is what is diagrammatically occurring in the above figure. In this case, it could be Internet access to a remote machine with a description language that permits access to semantics that allow for self-modification to penetrate to the next lower-level.

It is important to understand that the semantics for self-modification are non-trivial. *Semantics have to exist for the process of self-modification to break through these barriers.* The process is not magic. It simply does not have an ability to penetrate a description language barrier without the relevant methods at its disposal.

It is trivial to construct such curtailed worlds. It is done through an interpreter that simply *lacks* the semantics that would allow descriptions to manipulate APIs, I/O, device drivers, and so forth. Any programs under such a description language would be shut into a self-contained domain, a world with no possible escape without external aid. This is in stark contrast to one in which the semantics exist but have been guarded through various security measures.

The lack of enabling semantics should not be taken as an inability for such a system to *communicate* outside of the locked in world, which might be possible through side-channel attacks, such as timing analysis [82]. This, however, could be mitigated by implementing a real-time constraint on computation, such that the host interpreter is computing at a uniform or sufficiently uncorrelated rate with respect to program complexity.

The above methods describe various levels of sandboxing, and must not be taken as an argument for self-security. A sandboxed system can be overcome through external means, flaws in the implementation, and soft-errors regardless of the presence or absence of semantics and no matter how well guarded. The reason this discussion is made here regarding enabling semantics is to draw parallels between the existence and non-existence of doorways for self-modification.

As discussed above, the greater the distance between an implementation and the analogue representation in the discrete physics, the poorer the efficiency. As such, self-modification is a means of increasing the efficiency of a system by accessing either better semantics or by approaching a closer correspondence to an optimal state of computation for the problem. While this may sound abysmally utilitarian, it can be the difference between a tractable and intractable solution; the actual ranges for decent real-time performance are actually small, and it can be extremely difficult to get complex computation down into such ranges, especially on general purpose hardware. The challenges are not always in thinking up a method to solve a problem, but in making it *fast* on available hardware.

4.7 Metamorphic Strong AI

A metamorphic program [83, 94] is capable of recognizing and manipulating its own description language(s). The parenthetic plural is due to the fact that a program description may have more than one representation in the form of compiled code; a description in the human-readable source form, which may or may not be available to the program, and the native machine code representation. It is the latter that is most pressing with respect to metamorphic programs, as it automatically grants the ability to recognize any other program that has been compiled for the same architecture.

These programs are distinct from self-hosted interpreters in that their primary purpose is not to facilitate or implement program descriptions but to implement a particular set of program features which may change over time. Such programs utilize self-modification as a *means*, not an end, for enhancing these features or to discover new ones entirely.

The advantage of viewing strong AI as metamorphic software is that it simplifies the analysis. From a security standpoint, we can focus on the description, how and when it changes, and what other descriptions it modifies. We do not necessarily have to understand the description or the program behavior it implements. This is tremendously powerful, as we can recognize and classify strong AI by their descriptions and the description languages they are able to recognize. This could be used to create a hierarchy of increasingly complex autonomous systems that are graded based on their ability to manipulate these description languages or acquire new ones.

If we admit a representation where a strong AI description or implementation includes what it learns as part of its total description, then we have a basis for dovetailing all of current studies in artificial intelligence into the framework of self-modifying systems. That is, self-modification need not be restricted to what is typically thought of as machine op-codes and low-level programming constructs. Recall that description languages may be induced where any consistent structure can be applied. That they can then be connected with the rest of computer science and mathematics is what lends to their beauty as a unifying basis. Thus, that an implementation is modifying itself does not necessarily imply that it is altering its architecture; it could be modifying its knowledge and memories, or, in a more complex case, unavoidably altering its computational substrata because there is no longer a distinction between discrete units of computation and units of storage.

Machine learning comes into play at the intersection between the acquisition of new description languages and the manipulation of the descriptions in self-modifying implementations. This generalizes the notion of machine learning to that of the adaptation and effective manipulation of descriptions in some set of description languages, with the breadth and quality of that set defining a measure of its raw intellectual capacity. This works so long as one is willing to admit a flexible interpretation of description languages; they can be applied to virtually any physical process or system, even those that are under-specified, such as the micro and macro features of non-verbal communication in humans, or the particular way a set of servos must apply forces for some specific robotic instrument, all of which may vary from implementation to implementation due to subtle imperfections and environmental effects. Each is a description language in that it specifies a code, a way of transforming representations, and, with that comes consistent structure, lest it would be incomprehensible or unpredictable.

This is, however, non-trivial, as the semantics for any new description languages would need to be known in advance or inferred from an existing set of implementation semantics. Without this, the implementation would have to fall back to associative approaches that rely on mining data and constructing very high-dimensional representations of what could have potentially been vastly simpler structures. A case in point would be an attempt at mining all the ways one could signal a greeting, with no semantics for greetings available. This would result in brute enumeration of any possible combination in auditory, visual, and other modalities. It would, of course, collapse to descriptions, with the modalities simply being description languages. The system would then have to construct n-grams representing chains of presumed independent events that build a massive state space of possible behaviors based on where one is without regard to where one has been [84, 85, 93].

The Markov property [86] just described is essentially the default of any system that lacks semantic capacity; it only has incidence at its disposal. This is the degenerate case of any narrow AI system, as it necessarily lacks the semantic faculties that a strong AI must have. This is not a weakness or fault in the foundations but a reflection in the choice of philosophy or epistemology in engineering practice. It’s a misreading or misapplication of how knowledge is acquired by ignoring the vast compression that signification represents. That is to say, signification is merely an externalization of associated semantics. This is not to be confused with tacit versus explicit knowledge [87]. There is a gulf between semantics and even tacit knowledge, with the semantics of an implementation presupposing any possible knowledge representation, be it tacit or explicit. This is completely overlooked in conventional learning algorithm construction, not even on the table for consideration; it simply doesn't exist in the conceptual space of the relevant literature.

How does this apply to self-modifying systems? In that it sets up a barrier that a metamorphic program may possibly overcome due to the fact that its ability to self-interpret is independent of its ability to be executed. This allows the possibility of acquiring the use of description language semantics that were not previously available to the language that created it. Such a feat is not *directly* possible through self-hosting interpreters, even meta-circular ones, as there is always a footprint or underlying implementation in the host description language which is required to recognize the structure of the description language being implemented. This is not the case with metamorphic programs, as they are capable of transcending architectures and instruction sets. Such programs would have the capacity, through trial-and-error, or, the relation of existing semantics, to test and probe for semantics in the new description languages in which they rewrite themselves. This process would necessarily be error-prone and slow, no matter how "intelligent" a system (slow, relative to direct application of existing knowledge).

The analysis of metamorphic programs thus admits a convergence with bioinformatics and systems biology, in that we can directly interpret these implementations through the same techniques used for detecting mutations, insertions, and deletions in genetic sequences. All we must do to begin is admit is that there are multiple description languages. We would use generalized algorithms, methods, and a kind of multi-dimensional cladistics to recognize the behavior of these systems across description language boundaries.

Moving away from the theoretical now and towards the practical, we arrive at the operation of metamorphic programs. Traditionally, this has been to evade detection through masking their unique fingerprint or signature, and to confuse heuristics used by anti-malware tools [88]. This is a battle between metamorphic malware and the countermeasures used to detect them, with the countermeasures on the losing side [89]. In the end, the only comparable defense will be to instrument strong AI that can anticipate and adapt to other metamorphic strong AI. Crucally, *all strong AI will essentially be metamorphic*.

To quickly recap:

- Self-modification has the potential to overcome description language barriers and affords opportunities for the potential acquisition of new semantics.
- Manipulation and acquisition of description languages determine the threat model of metamorphic AI.
- The acquisition of semantics is non-trivial, and, in its absence, degenerates knowledge acquisition to high-dimensional rote grammars from brute-force enumeration.
- Breaking the meta-stability of self-modification is not done through magic. It requires *enabling semantics* to shatter the language barriers that keep it there, e.g. the ability to perform rewrite, replicate, or generate operations.
- The execution of metamorphic programs is independent of their ability to self-interpret.
- Systems biology and cladistics can be used to categorize and classify metamorphic descriptions, which could lead to better understanding

and actionable knowledge.

So far, this section has primarily focused on the positive or neutral effects of self-modification, but it is in the threats that the metamorphic perspective comes into full perspective. Consider programs which have the ability to rewrite restricted AI implementations, turning them into fully or partially unrestricted versions or restructuring them entirely to carry out malicious intent. Instead of malware that disrupts a specialized system or network, you could have adaptive or generalized strong AI malware that spreads through our global communications infrastructure, destroying or subverting existing implementations that have become compromised. This is an eventuality that this analysis can help to at least understand and anticipate.

The above statements come with a serious qualifier: strong AI is *not* going to spontaneously develop a persona and a sense of survival *ex nihilo*, then suddenly seek to overcome our civilization. Rather, it will be mere human beings who will craft these implementations and then release malicious versions of strong AI into the world. It could be a person or group doing it for the "lulz" (enjoyment in exercising the power to destroy simply because they can) [90], or it could be a government or non-state actor. All of these will be serious threats to cybersecurity and to the societies that are impacted. This is not some nebulous dollar figure attached to lost productivity or hampering of business as usual. Metamorphic strong AI malware will be beyond human-level malware, and will have the potential capacity to act as a local beligerent to the system beyond mere replication and disruption; it provides intelligence as a payload, and is the ultimate form of cybersecurity warfare. They will be virtual agents behind the lines, capable of exploiting and adapting vulnerabilities that would be impossible to detect remotely. The sword cuts both ways, however, as it is also going to be the *optimal defense*.

Metamorphic strong AI would be capable of analyzing existing machine code for vulnerabilities that would be beyond the ability for human experts to *reasonably* parse and analyze at that level, even with directly access to the information. Used defensively, these programs would be benign versions of the same destructive metamorphic software described above; instead of triggering a replicate or rewrite phase, it would report that information for human intervention. And it is at this level that we will first see strong AI instrumented and used on a wide scale in the information and security sectors. This is perhaps the closest reason for the name of this book: to become AI secure is to utilize the most effective strong AI implementation to seek out and eliminate vulnerabilities that only they could reliably detect and overcome. One's system may be secure, but it may not be AI secure. This cleanly entails the essence of the threat model. We won't be dealing with just human minds, but automated processes and cognitive implementations that are effective at levels that are beyond our best and brightest. In such a case, a metamorphic analysis is central in a warfare based on quickly changing descriptions across many architectures.

To employ these artificial minds in our defense, however, we must understand the technical, philosophical, and ethical implications of machine consciousness.

References

1. C. E. Shannon, "A mathematical theory of communication," ACM SIGMOBILE Mobile Computing and Communications Review, vol. 5, no. 1, pp. 3–55, 2001.
2. M. J. Golay, Notes on digital coding, vol. 37. 1949.
3. R. W. Hamming, "Error detecting and error correcting codes," Bell System technical journal, vol. 29, no. 2, pp. 147–160, 1950.

4. T. Jiang, M. Li, B. Ravikumar, and K. W. Regan, “Formal grammars and languages,” in Algorithms and theory of computation handbook, 2010, pp. 20–20.

5. G. Rozenberg and A. Salomaa, Handbook of Formal Languages: Beyonds words, vol. 3. Springer Science & Business Media, 1997.

6. M. A. Harrison, Introduction to formal language theory. Addison-Wesley Longman Publishing Co., Inc., 1978.

7. N. Chomsky, Syntactic structures. Walter de Gruyter, 2002.

8. J. R. Searle, Chomsky’s revolution in linguistics, vol. 18. New York Review of Books, 1974.

9. S. Harnad, “The symbol grounding problem,” Physica D: Nonlinear Phenomena, vol. 42, no. 1, pp. 335–346, 1990.

10. J. R. Searle, “Minds, brains, and programs,” Behavioral and brain sciences, vol. 3, no. 03, pp. 417–424, 1980.

11. M. Davis, R. Sigal, and E. J. Weyuker, Computability, complexity, and languages: fundamentals of theoretical computer science. Academic Press, 1994.

12. N. Chomsky, Aspects of the Theory of Syntax, vol. 11. MIT press, 1969.

13. W. F. Gilreath and P. A. Laplante, “One Instruction Set Computing,” in Computer Architecture: A Minimalist Perspective, Springer, 2003, pp. 1–3.

14. T. L. Short, Peirce’s theory of signs. Cambridge University Press, 2007.

15. T. W. Pratt, M. V. Zelkowitz, and T. V. Gopal, Programming languages: design and implementation. Prentice-Hall Englewood Cliffs, 1984.

16. M. J. Gordon, Programming language theory and its implementation. Prentice-Hall International Englewood Cliffs, 1988.

17. B. Russell, Introduction to mathematical philosophy. Courier Corporation, 1993.

18. R. H. Baayen, P. Hendrix, and M. Ramscar, “Sidestepping the combinatorial explosion: Towards a processing model based on discriminative learning,” in Empirically examining parsimony and redundancy in usage-based models, LSA workshop, 2011.

19. I. Arnon and N. Snider, “More than words: Frequency effects for multi-word phrases,” Journal of Memory and Language, vol. 62, no. 1, pp. 67–82, 2010.

20. T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” arXiv preprint arXiv:1301.3781, 2013.

21. G. D. Plotkin, “A structural approach to operational semantics,” 1981.

22. B. C. Smith, “Procedural reflection in programming languages,” Massachusetts Institute of Technology, 1982.

23. C. Strachey, “The varieties of programming language,” in Algol-like Languages, Springer, 1997, pp. 51–64.

24. M. D. McIlroy, “Macro instruction extensions of compiler languages,” Communications of the ACM, vol. 3, no. 4, pp. 214–220, 1960.

25. R. Jones and I. Stewart, “Compilers and Interpreters,” in The Art of C

Programming, Springer, 1987, pp. 1–4.

26. S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller, “Static and dynamic program compilation by interpreter specialization,” Higher-Order and Symbolic Computation, vol. 13, no. 3, pp. 161–178, 2000.
27. J. Aycock, “A brief history of just-in-time,” ACM Computing Surveys (CSUR), vol. 35, no. 2, pp. 97–113, 2003.
28. T. Lindholm and F. Yellin, Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc., 1999.
29. J. Smith and R. Nair, Virtual machines: versatile platforms for systems and processes. Elsevier, 2005.
30. B. Venners, Inside the Java virtual machine. McGraw-Hill, Inc., 1996.
31. U. A. Force, “Analysis of the intel pentium’s ability to support a secure virtual machine monitor,” 2000.
32. T. Garfinkel and M. Rosenblum, “When Virtual Is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments.,” in HotOS, 2005.
33. T. Garfinkel, M. Rosenblum, and others, “A Virtual Machine Introspection Based Architecture for Intrusion Detection.,” in NDSS, 2003, vol. 3, pp. 191–206.
34. M. A. Schuette and J. P. Shen, “Processor control flow monitoring using signatured instruction streams,” Computers, IEEE Transactions on, vol. 100, no. 3, pp. 264–276, 1987.
35. S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, “Flexible hardware acceleration for instruction-grain program monitoring,” ACM SIGARCH Computer Architecture News, vol. 36, no. 3, pp. 377–388, 2008.
36. V. Nagarajan and R. Gupta, “Runtime monitoring on multicores via oases,” ACM SIGOPS Operating Systems Review, vol. 43, no. 2, pp. 15–24, 2009.
37. D. Lo and G. E. Suh, “Worst-case execution time analysis for parallel run-time monitoring,” in Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, 2012, pp. 421–429.
38. J. C. Martinez Santos, Y. Fei, and Z. J. Shi, “Static secure page allocation for light-weight dynamic information flow tracking,” in Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems, 2012, pp. 27–36.
39. M. Ganai, D. Lee, and A. Gupta, “DTAM: dynamic taint analysis of multi-threaded programs for relevancy,” in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, p. 46.
40. V. Karakostas, S. Tomic, O. Unsal, M. Nemirovsky, and A. Cristal, “Improving the energy efficiency of hardware-assisted watchpoint systems,” in Proceedings of the 50th Annual Design Automation Conference, 2013, p. 54.
41. P. Ferrie, “Attacks on more virtual machine emulators,” Symantec Technology Exchange, 2007.
42. M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, “Virtualization for high-performance computing,” ACM SIGOPS Operating Systems Review, vol. 40, no. 2, pp. 8–11, 2006.

43. J. Rose, R. J. Francis, D. Lewis, and P. Chow, “Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency,” Solid-State Circuits, IEEE Journal of, vol. 25, no. 5, pp. 1217–1225, 1990.

44. A. P. Chandrakasan and R. W. Brodersen, “Minimizing power consumption in digital CMOS circuits,” Proceedings of the IEEE, vol. 83, no. 4, pp. 498–523, 1995.

45. T. Kalganova and J. Miller, “Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness,” in Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on, 1999, pp. 54–63.

46. M. D. Matson and L. A. Glasser, “Macromodeling and optimization of digital MOS VLSI circuits,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 5, no. 4, pp. 659–678, 1986.

47. D. Jarvis, “The effects of interconnections on high-speed logic circuits,” Electronic Computers, IEEE Transactions on, no. 5, pp. 476–487, 1963.

48. M. Orshansky, L. Milor, P. Chen, K. Keutzer, and C. Hu, “Impact of spatial intrachip gate length variability on the performance of high-speed digital circuits,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 21, no. 5, pp. 544–553, 2002.

49. S. P. Levitan and D. M. Chiarulli, “Massively parallel processing: It’s Déjà Vu all over again,” in Proceedings of the 46th Annual Design Automation Conference, 2009, pp. 534–538.

50. J. L. Gustafson, “Reevaluating Amdahl’s law,” Communications of the ACM, vol. 31, no. 5, pp. 532–533, 1988.

51. L. Snyder, “Type architectures, shared memory, and the corollary of modest potential,” Annual review of computer science, vol. 1, no. 1, pp. 289–317, 1986.

52. G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in Proceedings of the April 18-20, 1967, spring joint computer conference, 1967, pp. 483–485.

53. K. Golshan, Physical design essentials: An ASIC design implementation perspective. Springer Science & Business Media, 2007.

54. K.-C. Wu and Y.-W. Tsai, “Structured ASIC, evolution or revolution?,” in Proceedings of the 2004 international symposium on Physical design, 2004, pp. 103–106.

55. E. Ahmed and J. Rose, “The effect of LUT and cluster size on deep-submicron FPGA performance and density,” Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 12, no. 3, pp. 288–298, 2004.

56. E. Monmasson and M. N. Cirstea, “FPGA design methodology for industrial control systems—A review,” Industrial Electronics, IEEE Transactions on, vol. 54, no. 4, pp. 1824–1842, 2007.

57. J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, “Features, design tools, and application domains of FPGAs,” Industrial Electronics, IEEE Transactions on, vol. 54, no. 4, pp. 1810–1823, 2007.

58. J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, “Design of homogeneous communication infrastructures for partially reconfigurable FPGAs,” in Proc. of the Int. Conf. on Engineering of

Reconfigurable Systems and Algorithms (ERSA'07), 2007.

59. M. Dyer, C. Plessl, and M. Platzner, "Partially reconfigurable cores for Xilinx Virtex," in Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, Springer, 2002, pp. 292–301.

60. R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin, and C. Kitchen, An overview of FPGAs and FPGA programming: Initial experiences at Daresbury. Council for the Central Laboratory of the Research Councils, 2006.

61. S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A tutorial," IEEE Design and Test of Computers, vol. 13, no. 2, pp. 42–57, 1996.

62. Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura, "A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication," in Proceedings of the 41st annual Design Automation Conference, 2004, pp. 299–304.

63. P. Rashinkar, P. Paterson, and L. Singh, System-on-a-chip verification: methodology and techniques. Springer Science & Business Media, 2001.

64. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: reuse and integration," Proceedings of the IEEE, vol. 94, no. 6, pp. 1050–1069, 2006.

65. D. Langen, J.-C. Niemann, M. Porrmann, H. Kalte, and U. Rückert, "Implementation of a RISC processor core for SoC designs–FPGA prototype vs. ASIC implementation," in Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC), 2002.

66. M. Ernst, S. Klupsch, O. Hauck, and S. A. Huss, "Rapid prototyping for hardware accelerated elliptic curve public-key cryptosystems," in Rapid System Prototyping, 12th International Workshop on, 2001., 2001, pp. 24–29.

67. U. Y. Ogras, R. Marculescu, H. G. Lee, P. Choudhary, D. Marculescu, M. Kaufman, and P. Nelson, "Challenges and promising results in NoC prototyping using FPGAs," Micro, IEEE, vol. 27, no. 5, pp. 86–95, 2007.

68. P. H. W. Leong, "Recent trends in FPGA architectures and applications," in Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on, 2008, pp. 137–141.

69. F. J. Buckley, "Implementing configuration management. Hardware, software, and firmware," Los Alamitos, CA: IEEE Computer Society Press and Piscataway, NJ: IEEE Press,| c1996, 2nd ed., vol. 1, 1996.

70. T. Huffmire, Handbook of FPGA design security. Springer Science & Business Media, 2010.

71. T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. D. Nguyen, and C. Irvine, "Managing security in FPGA-based embedded systems," DTIC Document, 2008.

72. S. Drimer, "Volatile FPGA design security–a survey," IEEE Computer Society Annual Volume, pp. 292–297, 2008.

73. S. Trimberger, "Trusted design in FPGAs," in Proceedings of the 44th annual Design Automation Conference, 2007, pp. 5–8.

74. P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Moderator-Ravi, "Security as a new dimension in embedded system design," in Proceedings of the 41st annual Design Automation Conference, 2004, pp. 753–760.

75. Z. Zhou, J. Fan, N. Zhang, and R. Xu, "Advance and development of computer firmware security research," ISIP, vol. 2009, pp. 258–262, 2009.

76. D. K. Nilsson and U. E. Larson, "Secure firmware updates over the air in intelligent vehicles," in Communications Workshops, 2008. ICC Workshops' 08. IEEE International Conference on, 2008, pp. 380–384.

77. F. Adelstein, M. Stillerman, and D. Kozen, "Malicious code detection for open firmware," in Computer Security Applications Conference, 2002. Proceedings. 18th Annual, 2002, pp. 403–412.

78. K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and others, "Experimental security analysis of a modern automobile," in Security and Privacy (SP), 2010 IEEE Symposium on, 2010, pp. 447–462.

79. F. Stajano and H. Isozaki, "Security issues for internet appliances," in Applications and the Internet (SAINT) Workshops, 2002. Proceedings. 2002 Symposium on, 2002, pp. 18–24.

80. K. De Volder and P. Steyaert, "Construction of the reflective tower based on open implementations," Technical Report vub-prog-tr-95-01, Programming Technology Lab, Vrije Universiteit Brussel, 1995.

81. H. Abelson and G. J. Sussman, "Structure and interpretation of computer programs," 1983.

82. P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in Advances in Cryptology—CRYPTO'96, 1996, pp. 104–113.

83. W. Wong and M. Stamp, "Hunting for metamorphic engines," Journal in Computer Virology, vol. 2, no. 3, pp. 211–229, 2006.

84. F. Jelinek, "Interpolated estimation of Markov source parameters from sparse data," Pattern recognition in practice, 1980.

85. M. J. Beal, Z. Ghahramani, and C. E. Rasmussen, "The infinite hidden Markov model," in Advances in neural information processing systems, 2001, pp. 577–584.

86. A. A. Markov, "The theory of algorithms," Am. Math. Soc. Transl., vol. 15, pp. 1–14, 1960.

87. H. Collins, Tacit and explicit knowledge. University of Chicago Press, 2010.

88. E. Konstantinou and S. Wolthusen, "Metamorphic virus: Analysis and detection," London: Royal Holloway, 2008.

89. A. Sharma and S. Sahay, "Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey," arXiv preprint arXiv:1406.7061, 2014.

90. G. Coleman, "Anonymous: From the Lulz to collective action," The new everyday: a media commons project, vol. 6, 2011.

91. N. Chomsky, "On certain formal properties of grammars," Information and control, vol. 2, no. 2, pp. 137–167, 1959.

92. N. Chomsky, "Three models for the description of language," Information Theory, IRE Transactions on, vol. 2, no. 3, pp. 113–124,

1956.

93. P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," Computational linguistics, vol. 18, no. 4, pp. 467–479, 1992.

94. P. Ször and P. Ferrie, "Hunting for metamorphic," in Virus Bulletin Conference, 2001.

▲ Return to Top